Using A2C Algorithm to Develop an Game-Playing AI Agent For Geometry Dash

Clayton Clark, Andrew Dao, Jonathan Kao
Cal Poly Pomona
3801 W Temple Ave,
Pomona, CA 91768

Abstract

Using reinforcement learning, we aimed to create a game-playing AI for the game Geometry Dash, a rhythm based platforming game. In this paper we describe the approach that we took in creating our game-playing agent and environment using the Pygame library in Python. We also go over the A2C reinforcement learning algorithm that we created using TensorFlow. Upon further testing and repeated plays of our game, the results of our experiments with the A2C algorithm on Geometry Dash shows that it might not be the best reinforcement learning algorithm to run for this type of game. Other AI algorithms like Deep-Q Learning might be better suited for the game-play style of Geometry Dash.

1. Introduction

Geometry Dash is a rhythm based plat-former game where the player controls the movement of a block and navigates their way through side-scrolling, music-based levels avoiding obstacles in an attempt to complete the level.

Our project focuses on creating an artificial intelligent agent that is able to complete any level of Geometry Dash. Using reinforcement learning we have our agent play through the level performing one of two actions, jump or not jump. We then evaluate the state resulting from the agent's action and reward the agent for progressing through the level and punish the agent for dying. The agent is then tasked to maximize the reward and eventually complete the level.

We chose this concept for our project because we wanted to incorporate an artificial intelligence model in a video game. We initially wanted to implement a genetic algorithm since we were intrigued by the genetic cars example shown in class, but realized that it might not be the right approach for this type of game. We transitioned our approach to reinforcement learning and in this paper we present our findings.

2. Related Work

Video games have been around since the early 1970s and the industry continues to grow and develop new ways to embark on adventure, develop skills, or challenge the mind. At the end of the day, video games are a form of entertainment to the user. In recent years, AI has become more prevalent in games because of its ability in imitating ”human-like” behavior. Players are always looking for more immersion when it comes to video games and including artificial intelligent non-player characters is an approach that many games are implementing. However, AI is not an all encompassing solution to ”human-like” behavior in video games. Computer Role Playing Games(CRPGs) are where AI has fallen into the deep end because of the vast number of choices that an AI agent has to pick from (Spronck et al. 2006). This is where adaptive game AI becomes necessary because it has the ability to ”adapt successfully to changing circumstances”(Spronck et al. 2006). This type of adaptive game AI is mainly used when massive decision trees are present. There is some problems that are divided into 4 categories:

- Speed
- Effectiveness
- Robustness
- Efficiency

These are problems because of how the AI will need to learn as the game is going on in real time. The computational power and rigidity is vital in adaptive AI functioning correctly(Spronck et al. 2006). The article goes into depth about this type of AI and how to go about implementing dynamic scripting to work toward an adaptive game AI.

AI becomes very useful in a game like checkers. You have a board of 8x8 size with chips diagonal to each other. At the beginning of the game, your choices are limited 7 different possible outcomes. As the game progresses, the number of possible board layouts exponentially grows. This is where machine learning and AI can become a useful tool in learning better strategies or have an opponent AI making ”intelligent moves”. A.L. Samuel has an article going over two major machine learning procedures to getting a working AI. The two main procedures were a Neural Net Approach that had behavior based on a ”reward based routine” and a ”highly organized network” that will learn only ”certain things” (Samuel 1959).

3. Background

3.1 Reinforcement Learning

Reinforcement learning is an area of machine learning involving an agent and an environment. The agent has no prior knowledge of the environment, but through trial and error the agent learns by interacting with the environment using a set of actions. The agent is rewarded for actions that move it towards the goal and is punished for actions that move it away. The goal of the agent is to maximize the total cumulative reward. The general features of a reinforcement learning problem include:

- Environment - The world in which the agent interacts with
- State - The current circumstance the agent is in
- Reward - A value that tells the agent what is a good decision and what is a bad decision
- Policy - A strategy to tell agent what action it should take in a given state
- Value - An estimation of how good it is for an agent to be in a given state.

(Bhatt 2016)

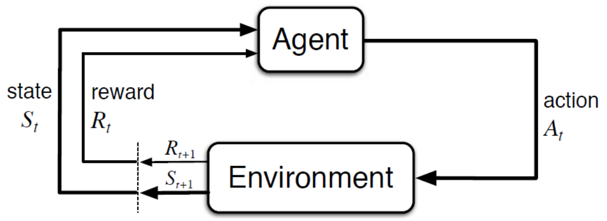

Figure 1: A visual representation of the agent and environment relationship.

In figure 1 we see how the agent and environment interact with each other in reinforcement learning approaches. Based on the state of the environment at a current time step and the reward associated with that state, the agent performs an action. With numerous time steps the agent will slowly learn which actions lead to more favorable states.

Reinforcement learning is common in creating artificial intelligent agents that are able to complete video games. Take the game Pac-Man for example, the goal of the agent, the Pac-Man, is to eat the food that is on the map and to avoid the ghosts. There are also special cases in which the Pac-Man is able to eat the ghosts. In the game of Pac-Man the environment is the map that the agent is in. The states include the current location of the agent in the map at the current time step. The agent is rewarded for eating food and punished if it gets killed by a ghost. The agent must maintain a balance of exploration and performing previously learned actions in order to learn the optimal policy and avoid running into a local maximum.

3.1.1 Advantage Actor Critic Advantage Actor Critic (A2C) is a reinforcement learning algorithm that we utilized for this project. It is a combination of two types of reinforcement learning algorithms, policy based and value based. A2C consists of an actor network and an critic network that work together to solve a problem. The actor network maps each state to a corresponding action and outputs a probability distribution corresponding to each action. The critic network evaluates the quality of a given input state and assigns it a Q-value. The Q-value numerically represents the quality of a given state. The A2C algorithm also features an advantage function that is calculated by taking the predicted value of all future rewards from the current state S (TD target) produced by the critic network and subtracting the value of the current state S (V(S)). The advantage function tells us if an action is better or worse than what we expected.

$$TDTarget = R + \gamma * V(S')$$

$$Advantage = TDTarget - V(S)$$

If it is worse we want to disincentivize the actor from taking this action, conversely if it is better we want to incentivize the actor to take more of that action. We can use the advantage to adjust the weights of our actor network to encourage taking better actions. The benefit of using the advantage function is that it helps reduce variance and thereby decrease learning time.

4. Gameplay Framework

A common way to train reinforcement learning models on games is to use OpenAI Gym environments that provide the necessary input to algorithms like A2C. In order to implement the A2C reinforcement learning algorithm, a framework needed to be established to be able to play the game of Geometry Dash and have control over the environment and the player or agent in the game.

Unlike other games that can be found on OpenAI, Geometry Dash is not native to the library. With some searching online, we were able to find a functioning version of Geometry Dash that was coded in Python. Some of the logic and assets were used in our version of the game, but some refactoring was necessary to replicate the environment input and output of an OpenAI Gym environment. The main function of any OpenAI Gym environment is the step function. This function should take an action, emulate the action at that step in the game, and return a new state, reward, and terminal variable. The action is the different controls or actions of the agent playing the game. The state that is returned is some metric or data of the state of the game after the step and action. The terminal variable is a boolean function that tells us if that step caused the game to terminate for that episode of training. We wanted to create the Geometry Dash environment to be compatible with this framework. These three pieces of information can be then fed into both an actor and critic neural network that decides if the actor is making correct actions based off of the observation that is made and the reward that is received.

5. Method

Our approach to this problem changed over time while working on the project. The reward and observation system went through major changes. The framework stayed relatively the same, but we changed our A2C algorithm implementation due to performance issues.

5.1 Initial Stages

The goal of the agent is to complete the game of Geometry Dash without dying on the spikes or the walls presented in the levels. The agent needs an action space to know what actions it has to choose from in its environment. The action space that is passed into the algorithm is of size two, one for jump and zero for idle. The critic needs an observation space to be able to predict if the action that was just made in the step was beneficial to the goal or unfavorable. Our initial observation space was a numpy array of metrics including the players current grid position, the distance to the next spike relative to the x-axis, and the current percentage of the level that was complete.

With these metrics, the agent should have been able to learn how to navigate the environment space based on the reward that was received at each step. Our first reward system was very simple. The agent would be awarded a small positive value for each time step that it was alive and a larger negative reward if it came into contact with a wall or spike, resulting in a game over.

5.2 Updated Reward

After some testing and training of the model, the reward system was changed to only give a positive reward if it had passed a spike and give a negative reward it it died to a spike or a wall. This incentivized the agent to jump over any of the spikes that it detects to stay alive rather than just using time alive as a reward system.

5.3 Updated Observation

After training with the three different observation metrics that were talked about in section 5.1, we realized that this was not a successful approach. We thought a better observation would be an image of the play environment, making our agent play through the game similar to how a human would. This would allow the AI to see more detail of the environment that could be used to better predict when the agent should or should not take an action.

In order to do this, the image data had to be compressed and grey scaled due to the performance hit that our computer took when training the model. The image that was used for the observation was scaled down from 600 x 600 pixels to 120 x 120 pixels. The color was also stripped from the image because the RGB values add an extra dimension to the equation. The output image from the game was 600 x 600 x 3 that equates to 1,080,000 values that the game must process each step. By prepossessing the image, the input to the neural network was changed to 120x120x1, which cuts the total input values to 14,400. Figure 2 shows the process of this pre-processing.

The only major drawback to using this method is that is takes a very long time to process the image data so it could be taxing on computer resources and slow down the run time of the game. It also takes much longer to train because each frame must be rendered in order to take the image to view the state of the environment. With the old observation system of using calculated distances, the model was able to train much faster because it was not necessary to render the game. This is one of the drawbacks of using images as observation, but it can be beneficial to the model's success.

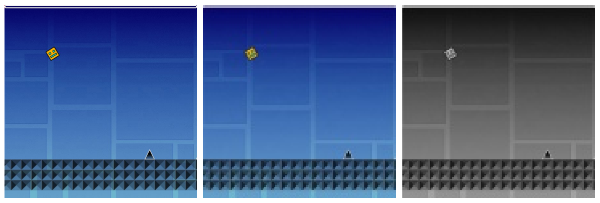

Figure 2: Pre-processing of Image Data for Neural Network Input

5.4 Training the Model

The actor and critic models needed to be trained over iterations of the game called episodes. Each episode is a play through of Geometry Dash up until the game ends in a a win or loss. In the beginning of training, the weights in the models are initialized to a random distribution and will be changed over time as episodes pass and weights are calculated and saved to the model. This means there needs to be some random actions made to progress and explore the environment space. A value called epsilon is set to 1 initially. A random number is picked ranging from 0 to 1 and compared to the epsilon value. If the random number is greater than the epsilon value, the program will choose to sample a random action that is not influenced by the learned policy values. This helps to keep the model from getting hard stuck on a local minimum and encourages exploring the environment and trying new things. As training progresses, the epsilon value is decremented by a fixed percentage of 0.25% after each training episode. Over the course of training, the model will begin to learn patterns from the environment and the rules of the game, and as epsilon decreases, the agent relies more on the policy that is learned to take actions.

6. Results

We ran into some challenges when training the AI and getting it to complete a level. At the end of a training cycle, the learned model would have 1 of the 2 following outcomes: It would learn to always want to jump, or it would learn to always want to idle. There was not much data that shows that it was varying the actions that it took at each given step. Our hypothesis is that the most of the training cycle is spent in the first tenth of the level and finds a way to always beat that portion but does not get to learn much at the later stages of the level because it gets there less frequently. We believe that a reinforcement learning algorithm like Deep Q-learning could have been a better fit for this game rather than the policy based A2C learning algorithm that we have tested.

Below, in Figure 3, is a graph that shows each of the episodes and their respective rewards at each episode.

In figure 3, notice that there is still an upward moving average at over 1200 training episodes but it is drastically hindered by a lack of learning the level. The maximum reward that any episode from 600 on to 1200 was around +400

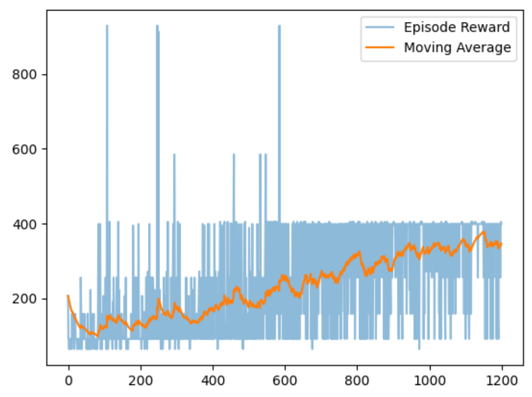

Figure 3: 3 Metric Observation Hard Map

reward. This is because the level that the AI was playing involved many consecutive jumping actions in a row to survive up until 1 spike in the level where the agent must idle for a brief moment before the next spike jump. It would get stuck in this loop of constantly jumping and mistiming the last jump before it was able to continue the level.

Below in Figure 4, we tried to train the model on a different level that involved much simpler jumps but they must be timed correctly in order to continue. This level forces the AI to learn how to idle before making a precise jump.

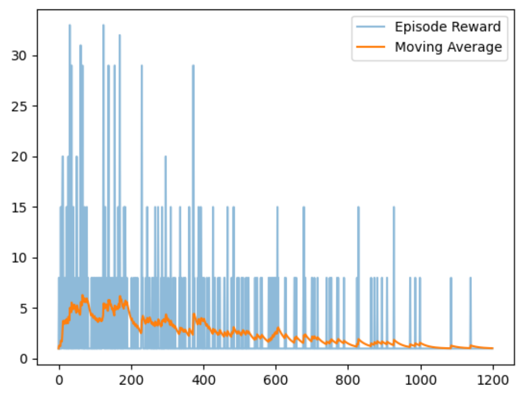

Figure 4: 3 Metric Observation Easier Map

The graph shows that the performance of the model was diminishing as it started to follow its learned policy. The AI decided to always jump at the start of the level, which would cause a collision with the first spike because idling was required at the start of the level followed by a timed jump. We figured that the reward system might need to be changed. A very small reward was added for continuing to idle at each step because a human player would know to time the jumps rather than continuously jump.

Below is the figure 5 that shows the average reward of each episode with new reward system.

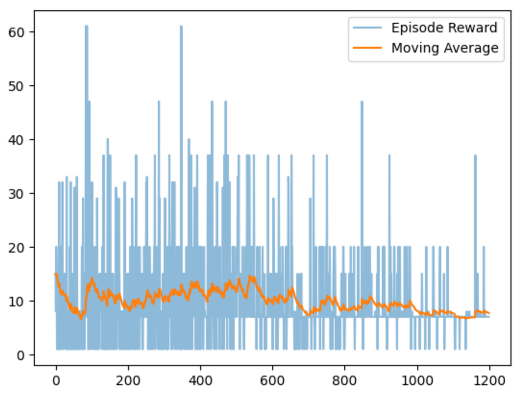

Figure 5: 3 Metric Observation Easier Map with New Reward for Being on the Round

We found that this reward system negatively impacted the way the agent learned because no matter the level that was played, by the end of training, it would always idle.

We reverted back to the original reward system of giving positive reward for getting passed a spike and giving negative reward for dying to a spike or a wall. The observation became the image of the game environment. The image was resized and grey scaled to increase the performance of the game and improve training time.

Below is the graph with our findings in Figure 6.

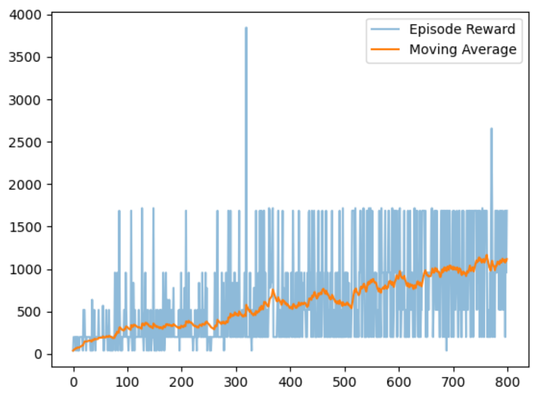

Figure 6: Using Prepossessed Images of Environment for Observation

The method was overall much better for training the model, but it was not flawless. The model was still getting stuck at certain parts of the level and not making good decisions on when to jump. The model was also very time consuming to train. With this method, only 800 episodes were used in the graph because the training time more than doubled that of having calculated metrics into the neural network. Our training time went from about 15 minutes for 1200 episodes to 45 minutes for 800 episodes using the images as observations. This is mostly due to having the render the game and play it out frame by frame rather than just calculating some values for observation and not having to render the game to train. We considered using services like Google Collab to accelerate this training process but quickly found out that Pygame, which is the main library the game is based on, was not supported on Collab and that the only work around was to process the game frame by frame using Matlab plotting tools within python that were very slow.

7. Future Work

There are a number of issues that we ran into during the implementation of this project and in the future we would like to address them. For one, we suspect that imagery data from just one singular frame of our environment may not be enough to serve as a useful observation. In an article about deep Q-networks it was mentioned that stacking consecutive frames of the environment could be used to provide a better observation (Mnih et al. 2015). This idea would help give the agent a better idea of the movement within our game. Take the example of the game Breakout, where the goal is to break as many bricks as possible by bouncing a ball between a paddle (the player) and the bricks. By just analyzing one frame of the game there is no sense of motion and it is difficult to discern whether the ball is moving towards the paddle or towards the bricks. In the future, we would like to test if frame stacking would be a viable observation that we could use to squeeze better performance from our agent. We also noticed that the agent's performance was quite sensitive to changes in the hyper-parameters of our model. In the future, we would like to perform extensive testing to see what sort of hyper-parameters will yield the best results. One issue with our implementation for this project is that for each level of Geometry Dash the agent and critic must be retrained. Compare this to other approaches where the agent might be able to generalize actions so that it can complete a level that it has never seen before in less training time. We would like to include something like this in our project in the future.

8. Conclusion

Using the Advantage Actor Critic reinforcement learning algorithm we were able to create an artificial intelligent agent that was able to play Geometry Dash. Although our approach did not yield the best results we still learned a lot from this project. Coming into this project none of us had experience in creating a game-playing AI so we had to learn how reinforcement learning and the A2C algorithm worked so we could implement it into our project. After learning about A2C conceptually, it was a whole different task of actually implementing it. We were unfamiliar with machine learning tools and libraries such as TensorFlow and had to slowly search through resources until we were able to implement the algorithm. In the end, we weren't able to achieve the results we were hoping for, but it was nice to see that our agent was still able to learn positive actions.

References

Bhatt, S. 2016. Reinforcement learning 101. https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292, (accessed: 11.20.2021).

García, S.; Fernández, A.; Luengo, J.; and Herrera, F. 2009. A study of statistical techniques and performance measures for genetics-based machine learning: accuracy and interpretability. *Soft Computing* 13(10):959.

Kachmar, O. T. 2016. Using genetic algorithms as a core gameplay mechanic. *Worcester Polytechnic Institute* 2–6.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.; Veness, J.; Bellemare, M.; Graves, A.; Riedmiller, M.; Fidjeland, A.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning.

Samuel, A. L. 1959. Some studies in machine learning using the game of checkers. *IBM Journal of research and development* 3(3):210–229.

Spronck, P.; Ponsen, M.; Sprinkhuizen-Kuyper, I.; and Postma, E. 2006. Adaptive game ai with dynamic scripting. *Machine Learning* 63(3):217–248.

Stephen Guy, I. K. 2015. Guide to anticipatory collision avoidance. *Game AI Pro 2* 195–201.

Wang, M. 2021. Advantage actor critic tutorial: mina2c. https://towardsdatascience.com/advantage-actor-critic-tutorial-mina2c-7a3249962fc8, (accessed: 11.20.2021).